A Syntactic Theory of Dynamic Binding

Luc Moreau*

University of Southampton

Abstract. Dynamic binding, which has always been associated with Lisp, is still semantically obscure to many. Although largely replaced by lexical scoping, not only does dynamic binding remain an interesting and expressive programming technique in specialised circumstances, but also it is a key notion in semantics. This paper presents a syntactic theory that enables the programmer to perform equational reasoning on programs using dynamic binding. The theory is proved to be sound and complete with respect to derivations allowed on programs in "dynamic-environment passing style". From this theory, we derive a sequential evaluation function in a context-rewriting system. Then, we exhibit the power and usefulness of dynamic binding in two different ways. First, we prove that dynamic binding adds expressiveness to a purely functional language. Second, we show that dynamic binding is an essential notion in semantics that can be used to define the semantics of exceptions. Afterwards, we further refine the evaluation function into the popular implementation strategy called deep binding. Finally, following the saying that deep binding is suitable for parallel evaluation, we present the parallel evaluation function of a future-based functional language extended with constructs for dynamic binding.

1 Introduction

Dynamic binding has traditionally been associated with Lisp dialects. It appeared in McCarthy's Lisp 1.0 [24] as a bug and became a feature in all succeeding implementations, like for instance MacLisp[2] [28], Gnu Emacs Lisp [23]. Even modern dialects of the language which favour lexical scoping provide some form of dynamic variables, with `special` declarations in Common Lisp [43], or even simulate dynamic binding by lexically-scoped variables as in MITScheme's `fluid-let` [18].

Lexical scope has now become the norm, not only in imperative languages, but also in functional languages such as Scheme [39], Common Lisp [43], Standard ML [26], or Haskell [21]. The scope of a name binding is the *text* where occurrences of this name refer to the binding. Lexical scoping imposes that a variable in an expression refers to the innermost lexically-enclosing construct declaring that variable. This rule implies that nested declarations follow a *block structure* organisation. On the contrary, the scope of a name is said to be *indefinite* [43] if references to it may occur anywhere in the program.

On the other hand, dynamic binding refers to a notion of dynamic extent. The *dynamic extent* of an expression is the lifetime of this expression, starting and ending when control enters and exits this expression. A dynamic binding is a binding which exists and can only be used during the dynamic extent of an expression. A dynamic variable refers to the latest active dynamic binding that exists for that variable [1]. The expression *dynamic scope* is convenient to refer to the indefinite scope of a variable with a dynamic extent [43].

Dynamic binding was initially defined by a meta-circular evaluator [24] and was later formalised by a denotational semantics by Gordon [15, 16]. It is also part of the

* This research was supported in part by EPSRC grant GR/K30773. Author's address: Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ. United Kingdom. E-mail: `L.Moreau@ecs.soton.ac.uk`.

[2] At least, the interpreted mode.

folklore that there exists a translation, the *dynamic-environment passing translation*, which translates programs using dynamic binding into programs using lexical binding only [36, p. 180]. Like the continuation-passing transform [35], the dynamic-passing translation adds an extra argument to each function, its dynamic environment, and every reference to a dynamic variable is translated into a lookup in the current dynamic environment.

The late eighties saw the apparition of "syntactic theories", a new semantic framework which allows equational reasoning on programs using non-functional features like first-class continuations and state [10, 11, 12, 44]. Those frameworks were later extended to take into account parallel evaluation [9, 14, 29, 30]. The purpose of this paper is to present a syntactic theory that allows the user to perform equational reasoning on programs using dynamic binding. Our contribution is fivefold.

First, from the dynamic-environment passing translation, we construct an inverse translation. Using Sabry and Felleisen's technique [40, 41], we derive a set of axioms and define a calculus, which we prove to be sound and complete with respect to the derivations accepted in dynamic-environment passing style (Section 3).

Second, we devise a sequential evaluation function, i.e. an algorithm, which we prove to return a value whenever the calculus does so. The evaluation function, which relies on a context-rewriting technique [11], is presented in Section 4.

Third, in order to strengthen our claim that dynamic binding is an expressive programming technique and a useful notion in semantics, we give a formal proof of its expressiveness and use it in the definition of exceptions. In Section 5, we define a relation of observational equivalence using the evaluation function, and we prove that dynamic binding adds expressiveness [8] to a purely functional programming language, by establishing that dynamic binding cannot be macro-expressed in the call-by-value lambda-calculus. In Section 6, we use dynamic binding as a semantic primitive to formalise two different models of exceptions: non-resumable exceptions as in ML [26] and resumable ones as in Common Lisp [43, 34].

Fourth, we refine our evaluation function in the strategy called *deep binding*, which facilitates the creation and restoration of dynamic environments (Section 7).

Fifth, we extend our framework to parallel evaluation, based on the future construct [14, 17, 30]. In Section 8, we define a parallel evaluation function which also relies on the deep binding technique.

Before deriving our calculus, we further motivate our work by describing three broad categories of use of dynamic binding: conciseness, control delimiters, and distributed computing. Let us insist here and now that our purpose is *not* to denigrate the qualities of lexical binding, which is the essence of abstraction by its block structure organisation, but to present a syntactic theory that allows equational reasoning on dynamic binding, to claim that dynamic binding is an expressive programming technique if used in a sensible manner, and to show that dynamic binding can elegantly be used to define semantics of other constructs. Let us note that dynamic binding is found not only in Lisp but also in TeX [22], Perl [45], and Unix^{TM} shells.

2 Practical Uses of Dynamic Binding

2.1 Conciseness

A typical use of dynamic binding is a printing routine `print-number` which requires the basis in which the numbers should be displayed. One solution would be to pass an explicit argument to each call to `print-number`. However, repeating such a programming pattern across the whole program is the source of programming mistakes. In addition, this solution is not scalable, because if later we require the `print-number` routine to take an additional parameter indicating in which font numbers should be displayed, we would have to modify the whole program.

Scheme I/O functions take an *optional* input/output port. The procedures `with-input-from-file` and `with-output-to-file` [39] simulate dynamic binding for these parameters.

Gnu Emacs [23] is an example of large program using dynamic variables. It contains dynamic variables for the current buffer, the current window, the current cursor position, which avoid to pass these parameters to all the functions that refer to them.

These examples illustrate Felleisen's conciseness conjecture [8], according to which sensible use of expressive programming constructs can reduce programming patterns in programs. In order to strengthen this observation, we prove that dynamic binding actually adds expressiveness to a purely functional language in Section 5.

2.2 Control Delimiters

Even though Standard ML [26] is a lexically-scoped language, raised exceptions are caught by the latest active handler. Usually, programmers install exception handlers for the duration of an expression, i.e. the handler is dynamically bound during the extent of the expression. MacLisp [28] and Common Lisp [43] `catch` and `throw`, Eulisp `let/cc` [34] are other examples of exception-like control operators with a dynamic extent. More generally, control delimiters are used to create partial continuations, whose different semantics tolerate various degrees of dynamicness [5, 20, 31, 38, 42].

2.3 Parallelism and Distribution

Parallelism and distribution are usually considered as a possible mean of increasing the speed of programs execution. However, another motivation for distribution, exacerbated by the ubiquitous WWW, is the quest for new resources: a computation has to migrate from a site s_1 to a another site s_2, because s_2 holds a resource that is not accessible from s_1. For our explanatory purpose, we consider a simple resource which is the name of a computer. There are several solutions to model the name of the running host in a language; the last one only is entirely satisfactory.

(*i*) A lexical variable `hostname` could be bound to the name of the computer whenever a process is created. Unfortunately, this variable, which may be closed in a closure, will always return the same value, even though it is evaluated on a different site.

(*ii*) A primitive `(hostname)`, defined as a function of its arguments only (by δ in [35]), cannot return different values in different contexts, unless it is defined as a non-deterministic function, which would prevent equational reasoning.

(*iii*) A special form `(hostname)` could satisfy our goal, but it is in contradiction with the minimalist philosophy of Scheme, which avoids adding unnecessary special forms. Furthermore, as we would have to define such a special form for every resource, it would be natural to abstract them into a unique special form, parameterised by the resource name: this introduces a new name space, which is exactly what dynamic binding offers.

(*iv*) Our solution is to dynamically bind a variable `hostname` with the name of the computer at process-creation time. Every occurrence of such a variable would refer to the latest active binding for the variable.

Besides, control of tasks in a parallel/distributed setting usually relies on a notion of dynamic extent: sponsors [33, 37] allow the programmer to control hierarchies of tasks.

3 A Calculus of Dynamic Binding

Figure 1 displays the syntax of Λ_u, the language accessible to the end user. Let us observe that the purpose of Λ_u is to capture the *essence* of dynamic variables and not to propose a new *syntax* for them.The language Λ_u is based on two disjoint sets of variables: the *dynamic* and *static* (or *lexical*) variables. As a consequence, the programmer can choose between lexical abstractions $\lambda x_s.M$ which lexically bind their parameter when applied, or dynamic abstractions $\lambda x_d.M$, which dynamically bind their parameter. The former represent regular abstractions of the λ-calculus [3], while the latter model constructs like Common Lisp abstractions with special variables [43], or `dynamic-scope` [6].

$$\begin{array}{llll}
M \in \Lambda_u & ::= & V \mid x_d \mid (M\ M) & (\mathit{Term}) \\
V \in \mathit{Value}_u & ::= & x_s \mid (\lambda x_s.M) \mid (\lambda x_d.M) & (\mathit{Value}) \\
x_s \in \mathit{SVar} & = & \{x_{s0}, x_{s1}, \ldots\} & (\mathit{Static\ Variable}) \\
x_d \in \mathit{DVar} & = & \{x_{d0}, x_{d1}, \ldots\} & (\mathit{Dynamic\ Variable})
\end{array}$$

Fig. 1. The User Language Λ_u

It is of paramount importance to clearly state the naming conventions that we adopt for such a language. Following Barendregt [3], we consider terms that are equal up to the renaming of their *bound static* variables as equivalent. On the contrary, two terms that differ by their dynamic variables are *not* considered as equivalent.

$$\begin{array}{rll}
\mathcal{D}[\![(M_1\ M_2), E]\!] & = (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_2, E]\!])\ \mathcal{D}[\![M_1, E]\!] & y_1 \notin FV(\mathcal{D}[\![M_2, E]\!]) \\
\mathcal{D}[\![\lambda x_d.M, E]\!] & = \lambda\langle e, y\rangle.\ \mathcal{D}[\![M, (\text{extend } e\ x_d\ y)]\!] \quad y \notin FV(M) & y_2 \notin FV(E) \\
\mathcal{D}[\![\lambda x_s.M, E]\!] & = \lambda\langle e, x_s\rangle.\ \mathcal{D}[\![M, e]\!] & \\
\mathcal{D}[\![x_d, E]\!] & = (\text{lookup } x_d\ E) & \\
\mathcal{D}[\![x_s, E]\!] & = x_s & \\
\mathcal{D}[\![(\mathsf{dlet}\ \delta\ M), E]\!] & = \mathcal{D}[\![M, \mathcal{B}[\![\delta, E]\!]]\!] & \\
\mathcal{B}[\![(), E]\!] & = E & \\
\mathcal{B}[\![\delta\ \S\ ((x_d\ V)), E]\!] & = (\text{extend } \mathcal{B}[\![\delta, E]\!]\ x_d\ \mathcal{D}[\![V, e]\!]) &
\end{array}$$

Fig. 2. Dynamic-Environment Passing Transform $\mathcal{D}$

In Figure 2, the *dynamic-environment passing translation*, which we call $\mathcal{D}$, is a program transformation that maps programs of Λ_u into the target language $deps(\Lambda_d)$, an extended call-by-value λ-calculus based on lexical variables only (Figure 3). Intuitively, each abstraction (static or dynamic) of Λ_u is translated by $\mathcal{D}$ into an abstraction taking an extra dynamic environment in argument; the target language contains a variable e which denotes an unknown environment. As a result, the application protocol in the target language is changed accordingly: operator values are applied to pairs. In the translation of the application, the dynamic environment E is used in the translations of the operator and operand, and is also passed in argument to the operator. Dynamic abstractions are translated into abstractions which extend the dynamic environment. Each dynamic variable is translated into a lookup for the corresponding constant in the current dynamic environment.

The source language of $\mathcal{D}$ extends Λ_u with a dlet construct, (dlet $((x_{d1}\ V_1)\ \ldots)\ M$), which stands for "dynamic let". Such a construct, inaccessible to the programmer, is used internally by the system to model the bindings of dynamic variables x_{di} to values V_i. The syntax of the input language, called Λ_d, appears in Figure 5. Binding lists are defined with the concatenation operator §, satisfying the following property.

$$((x_{d1}\ V_1)\ \ldots\ (x_{dn}\ V_n))\ \S\ ((x_{dn+1}\ V_{n+1})\ \ldots) = ((x_{d1}\ V_1)\ \ldots\ (x_{dn}\ V_n)\ (x_{dn+1}\ V_{n+1})\ \ldots)$$

Evaluation in the target language is based on the set of axioms displayed in the second part of Figure 3. Applications of binary abstractions require a double β_v-reduction as modelled by rule $(\beta_v^\times)$, and environment lookup is implemented by (lk_1) and (lk_2).

Following Sabry and Felleisen, our purpose in the rest of this Section is to derive the set of axioms that can perform on terms of Λ_d the reductions allowed on terms of

The Language *deps*(Λ_d)**:**

$$\begin{array}{rcll}
P & ::= & W \mid (W\ \langle E, W\rangle) \mid (\text{lookup}\ x_d\ E) \mid (\lambda y.P)P & (\mathit{Term}) \\
W & ::= & x_s \mid y \mid \lambda\langle e, y\rangle.P & (\mathit{Value}) \\
E & ::= & e \mid (\text{extend}\ E\ x_d\ W) \mid () & (\mathit{Dynamic\ Environment}) \\
 & & \text{e} & (\mathit{Unknown\ Env.\ Variable}) \\
x_s, y \in \mathit{SVars} & = & \{x_{s0}, x_{s1}, \ldots\} \cup \{y_0, \ldots\} & (\mathit{Static\ Variable}) \\
x_d \in \mathit{DConst} & = & \{x_{d0}, x_{d1}, \ldots\} & (\mathit{Dynamic\ Identifier})
\end{array}$$

Axioms:

$$\begin{array}{cr}
(\lambda\langle e, y\rangle.P)\langle E, W\rangle = P\{E/e\}\{W/y\} & (\beta_v^\times) \\
(\lambda y.P)W = P\{W/y\} & (\beta_v) \\
(\text{lookup}\ x_d\ (\text{extend}\ E\ x_d\ W)) = W & (lk_1) \\
(\text{lookup}\ x_d\ (\text{extend}\ E\ x_{d1}\ W)) = (\text{lookup}\ x_d\ E) \quad \text{if}\ x_{d1} \neq x_d & (lk_2) \\
(\lambda\langle e, y\rangle.W\langle e, y\rangle) = W\ \text{if}\ e, y \notin FV(W) & (\eta_v^\times)
\end{array}$$

Fig. 3. Syntax and Axioms of the *deps*(λ_d)-calculus

$$\begin{array}{rcl}
\mathcal{D}^{-1}[\![W_1\ \langle E, W_2\rangle]\!] & = & (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ (\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!])) \\
\mathcal{D}^{-1}[\![(\text{lookup}\ x_d\ E)]\!] & = & (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ x_d) \\
\mathcal{D}^{-1}[\![(\lambda y.P_1)\ P_2]\!] & = & (\lambda y.\mathcal{D}^{-1}[\![P_1]\!])\ \mathcal{D}^{-1}[\![P_2]\!] \\
\mathcal{D}^{-1}[\![(\lambda\langle e, y\rangle.P)]\!] & = & \lambda y.\mathcal{D}^{-1}[\![P]\!] \\
\mathcal{D}^{-1}[\![(\lambda\langle e, x_s\rangle.P)]\!] & = & \lambda x_s.\mathcal{D}^{-1}[\![P]\!] \\
\mathcal{D}^{-1}[\![x_s]\!] & = & x_s \\
\mathcal{B}^{-1}[\![e]\!] & = & () \\
\mathcal{B}^{-1}[\![(\text{extend}\ E\ x_d\ W)]\!] & = & (\mathcal{B}^{-1}[\![E]\!]\ \S\ ((x_d\ \mathcal{D}^{-1}[\![W]\!])))
\end{array}$$

Fig. 4. The Inverse Dynamic-Environment Passing Transform $\mathcal{D}^{-1}$

deps(Λ_d). More precisely, we want to define a calculus on Λ_d that *equationally corresponds* to the calculus on *deps*(Λ_d). The following definition of equational correspondence is taken verbatim from [40].

Definition 1 (Equational Correspondence) Let $\mathcal{R}$ and $\mathcal{G}$ be two languages with calculi $\lambda X_{\mathcal{R}}$ and $\lambda X_{\mathcal{G}}$. Also let $f : \mathcal{R} \to \mathcal{G}$ be a translation from $\mathcal{R}$ to $\mathcal{G}$, and $h : \mathcal{G} \to \mathcal{R}$ be a translation from $\mathcal{G}$ to $\mathcal{R}$. Finally let $r, r_1, r_2 \in \mathcal{R}$ and $g, g_1, g_2 \in \mathcal{G}$. Then the calculus $\lambda X_{\mathcal{R}}$ *equationally corresponds* to the calculus $\lambda X_{\mathcal{G}}$ if the following four conditions hold:

1. $\lambda X_{\mathcal{R}} \vdash r = (h \circ f)(r)$.
2. $\lambda X_{\mathcal{G}} \vdash g = (f \circ h)(g)$.
3. $\lambda X_{\mathcal{R}} \vdash r_1 = r_2$ if and only if $\lambda X_{\mathcal{G}} \vdash f(r_1) = f(r_2)$.
4. $\lambda X_{\mathcal{G}} \vdash g_1 = g_2$ if and only if $\lambda X_{\mathcal{R}} \vdash h(g_1) = h(g_2)$.

□

Figure 4 contains an inverse dynamic-environment passing transform mapping terms of *deps*(Λ_d) into terms of Λ_d. The first case is worth explaining: a term $(W_1\ \langle E, W_2\rangle)$ represents the application of an operator value W_1 on a pair dynamic environment E and operand value W_2; its inverse translation is the application of the inverse translations of W_1 and W_2, in the scope of a dlet with the inverse translation of E. For the following cases, the inverse translation removes the environment argument added to abstractions, and translates any occurrence of a dynamic environment into a dlet-expression.

State Space:

$$\begin{array}{lcll}
M \in \Lambda_d & ::= & V \mid x_d \mid (M\ M) \mid (\mathsf{dlet}\ \delta\ M) & (\mathit{Term}) \\
V \in \mathit{Value}_d & ::= & x_s \mid y \mid (\lambda x_s.M) \mid (\lambda x_d.M) & (\mathit{Value}) \\
\delta \in \mathit{Bind}_d & ::= & () \mid \delta \ \S\ ((x_d\ V)) & (\mathit{binding\ list}) \\
x_s, y \in \mathit{SVar} & ::= & \{x_{s0}, x_{s1}, \ldots\} \cup \{y_0, \ldots\} & (\mathit{Static\ Variable}) \\
x_d \in \mathit{DVar} & ::= & \{x_{d0}, x_{d1}, \ldots\} & (\mathit{Dynamic\ Variable})
\end{array}$$

Primary Axioms:

$$\begin{array}{rcll}
(\lambda x_s.M)\ V & = & M\{V/x_s\} & (\beta_v) \\
\lambda x_d.M & = & \lambda y.(\mathsf{dlet}\ ((x_d\ y))\ M) \quad \text{if } y \notin FV(M) & (\mathsf{dlet}\ \mathit{intro}) \\
(\mathsf{dlet}\ \delta\ ((\lambda y.M_1)\ M_2)) & = & (\lambda y.(\mathsf{dlet}\ \delta\ M_1))\ (\mathsf{dlet}\ \delta\ M_2) \quad \text{if } y \notin FV(\delta) & (\mathsf{dlet}\ \mathit{propagate}) \\
(\mathsf{dlet}\ \delta_1\ (\mathsf{dlet}\ \delta_2\ M)) & = & (\mathsf{dlet}\ (\delta_1\ \S\ \delta_2)\ M) & (\mathsf{dlet}\ \mathit{merge}) \\
(\mathsf{dlet}\ \delta\ V) & = & V & (\mathsf{dlet}\ \mathit{elim\ 1}) \\
(\mathsf{dlet}\ ()\ M) & = & M & (\mathsf{dlet}\ \mathit{elim\ 2}) \\
(\mathsf{dlet}\ (\delta\ \S\ ((x_d\ V)))\ x_d) & = & (\mathsf{dlet}\ (\delta\ \S\ ((x_d\ V)))\ V) & (\mathit{lookup\ 1}) \\
(\mathsf{dlet}\ (\delta\ \S\ ((x_{d1}\ V)))\ x_d) & = & (\mathsf{dlet}\ \delta\ x_d) \quad \text{if } x_{d1} \neq x_d & (\mathit{lookup\ 2}) \\
(\lambda x.x\ M_2)M_1 & = & (M_1\ M_2) \quad \text{if } x \notin FV(M_2) & (\beta'_\Omega) \\
(\lambda x.V\ x) & = & V \quad \text{if } x \notin FV(V) & (\eta_v)
\end{array}$$

Derived Axioms:

$$\begin{array}{rcll}
(\lambda x_d.M)\ V & = & (\mathsf{dlet}\ ((x_d\ V))\ M) & (\mathsf{dlet}\ \mathit{intro}') \\
(\mathsf{dlet}\ \delta\ (M_1\ M_2)) & = & (\lambda y_1.(\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M_2))\ (\mathsf{dlet}\ \delta\ M_1) & (\mathsf{dlet}\ \mathit{propagate}')
\end{array}$$

Compatibility

$$M_1 = M_2 \Rightarrow \begin{cases} (M_1\ M) = (M_2\ M) & (\lambda x_s.M_1) = (\lambda x_s.M_2) \\ (M\ M_1) = (M\ M_2) & (\lambda x_d.M_1) = (\lambda x_d.M_2) \\ & (\mathsf{dlet}\ \delta\ M_1) = (\mathsf{dlet}\ \delta\ M_2) \end{cases}$$

Fig. 5. Syntax and Axioms of the λ_d-calculus

If we apply the dynamic-environment passing transform $\mathcal{D}$ to a term of Λ_d, and immediately translate the result back to Λ_d by $\mathcal{D}^{-1}$, we find the first six primary axioms of Figure 5. For explanatory purpose, we prefer to present the derived axioms (dlet *intro'*) and (*dlet propagate'*). The axiom (dlet *intro'*) is the counterpart of (β_v) for dynamic abstraction: applying a dynamic abstraction on a value V creates a dlet-construct that dynamically binds the parameter to the argument V and that has the same body as the abstraction. Rule (dlet *propagate'*), rewritten below using the syntactic sugar let, tells us how to transform an application appearing inside the scope of a dlet.

$$(\mathsf{dlet}\ \delta\ (M_1\ M_2)) = (\mathsf{let}\ (y_1\ (\mathsf{dlet}\ \delta\ M_1))\ \ (\mathsf{let}\ (y_2\ (\mathsf{dlet}\ \delta\ M_2))\ \ (\mathsf{dlet}\ \delta\ (y_1\ y_2))))$$

The operator and the operand can each separately be evaluated inside the scope of the same dynamic environment, and the application of the operator value on the operand value also appears inside the scope of the same dynamic environment. The interpretation of (dlet *merge*), (dlet *elim 1*), (dlet *elim 2*) is straightforward.

We can establish the following properties concerning the composition of $\mathcal{D}$ and $\mathcal{D}^{-1}$:

Lemma 2 For any term $M \in \Lambda_d$, any value $V \in \mathit{Value}_d$, any list of bindings $\delta_1 \in$

$Bind_d$, for any environment $E \in deps(\Lambda_d)$, let $\delta = \mathcal{B}^{-1}[\![E]\!]$, we have:

$$\lambda_d \vdash (\mathsf{dlet}\ \delta\ M) = \mathcal{D}^{-1}[\![\mathcal{D}[\![M,E]\!]\!]\ (1) \qquad \lambda_d \vdash \delta\ \S\ \delta_1 = \mathcal{B}^{-1}[\![\mathcal{B}[\![\delta_1, E]\!]\!]\ (3)$$
$$\lambda_d \vdash V = \mathcal{D}^{-1}[\![\mathcal{D}[\![V,E]\!]\!] \qquad (2)$$

□

Lemma 3 For any term $P \in deps(\Lambda_d)$, any value $W \in deps(Value_d)$, any dynamic environments $E, E_1 \in deps(\Lambda_d)$, we have:

$$deps(\lambda_d) \vdash \mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] = P\{E/e\}\ (1) \qquad deps(\lambda_d) \vdash \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!] = E_1\{E/e\}\ (3)$$
$$deps(\lambda_d) \vdash \mathcal{D}[\![\mathcal{D}^{-1}[\![W]\!], E]\!] = W \qquad (2)$$

□

Now, by applying the inverse translation $\mathcal{D}^{-1}$ to each axiom of $deps(\lambda_d)$, we obtain the four last primary axioms of Figure 5. Rules (*lookup 1*) and (*lookup 2*) are the immediate correspondent of (lk_1) and (lk_2) in $deps(\lambda_d)$, while (β'_Ω) and (η_v) were axioms discovered by Sabry and Felleisen in applying the same technique to calculi for continuations and assignments [40].

The intuition of the set of axioms of λ_d can be explained as follows. In the absence of dynamic abstractions, λ_d behaves as the call-by-value λ-calculus. Whenever a dynamic abstraction is applied, a dlet construct is created. Rule (dlet *propagate'*) propagates the dlet to the leaves of the syntax tree, and replaces each occurrence of a dynamic variable by its value in the dynamic environment by (*lookup 1*) and (*lookup 2*). Rule (dlet *propagate'*) also guarantees that the dynamic binding remains accessible during the extent of the application of the dynamic abstraction, i.e. until it is deleted by (dlet *elim 1*). Let us also observe here and now that parallel evaluation is possible because the dynamic environment is duplicated for the operator and the operand, and both can be reduced independently. This property will be used in Section 8 to define a parallel evaluation function. We obtain the following *soundness* and *completeness* results:

Lemma 4 (Soundness) For any terms $M_1, M_2 \in \Lambda_d$, such that $\lambda_d \vdash M_1 = M_2$, and for any $E \in deps(\Lambda_d)$, we have that: $deps(\lambda_d) \vdash \mathcal{D}[\![M_1, E]\!] = \mathcal{D}[\![M_2, E]\!]$. □

Lemma 5 (Completeness) For any terms $P_1, P_2 \in deps(\Lambda_d)$, such that $deps(\lambda_d) \vdash P_1 = P_2$, we have that: $\lambda_d \vdash \mathcal{D}^{-1}[\![P_1]\!] = \mathcal{D}^{-1}[\![P_2]\!]$ □

The following Theorem is a consequence of Lemmas 2 to 5.

Theorem 1 *The calculus λ_d equationally corresponds to the calculus $deps(\lambda_d)$.* □

Within the calculus, we can define a partial evaluation relation: the value of a program M is V if we can prove that M equals V in the calculus.

Definition 6 (eval$_c$) For any program $M \in \Lambda_u^0$, eval$_c(M) = V$ if $\lambda_d \vdash M = V$. □

This definition does not give us an algorithm, but it states the specification that must be satisfied by any evaluation procedure. The purpose of the next Section is to define such a procedure.

4 Sequential Evaluation

The sequential evaluation function is defined in Figure 6. It relies on a notion of evaluation context [11]: an evaluation context $\mathcal{E}$ is a term with a "hole", [ ], in place of the next subterm to evaluate. We use the notation $\mathcal{E}[M]$ to denote the term obtained by placing M inside the hole of the context $\mathcal{E}$. Four transition rules only are necessary: (dlet *intro*) and (dlet *elim*) are derived from the λ_d-calculus. Rule (*lookup*) is a replacement for (dlet *propagate*), (dlet *merge*), (dlet *lookup 1*), and (dlet *lookup 2*) of the λ_d-calculus.

State Space:

$$\begin{array}{lcll}
M \in \Lambda_d & ::= & V \mid x_d \mid (M\ M) \mid (\mathsf{dlet}\ (x_d\ V)\ M) & (\mathit{Term}) \\
V \in \mathit{Value}_d & ::= & x_s \mid (\lambda x_s.M) \mid (\lambda x_d.M) & (\mathit{Value}) \\
x_s \in \mathit{SVar} & = & \{x_{s0}, x_{s1}, \ldots\} & (\mathit{Static\ Variable}) \\
x_d \in \mathit{DVar} & = & \{x_{d0}, x_{d1}, \ldots\} & (\mathit{Dynamic\ Variable}) \\
\mathcal{E} \in \mathit{EvCon}_d & ::= & [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{dlet}\ (x_d\ V)\ \mathcal{E}) & (\mathit{Evaluation\ Context})
\end{array}$$

Transition Rules:

$$\begin{array}{rll}
\mathcal{E}[(\lambda x_s.M)\ V] \mapsto_d \mathcal{E}[M\{V/x_s\}] & & (\beta_v) \\
\mathcal{E}[(\lambda x_d.M)\ V] \mapsto_d \mathcal{E}[(\mathsf{dlet}\ (x_d\ V)\ M)] & & (\mathsf{dlet}\ \mathit{intro}) \\
\mathcal{E}[(\mathsf{dlet}\ (x_d\ V)\ \mathcal{E}_1[x_d])] \mapsto_d \mathcal{E}[(\mathsf{dlet}\ (x_d\ V)\ \mathcal{E}_1[V])] & \text{if } x_d \notin DBV(\mathcal{E}_1) & (\mathit{lookup}) \\
\mathcal{E}[(\mathsf{dlet}\ (x_d\ V)\ V')] \mapsto_d \mathcal{E}[V'] & & (\mathsf{dlet}\ \mathit{elim})
\end{array}$$

Evaluation Function:

$$\text{For any program } M \in \Lambda_u^0,\ \mathrm{eval}_d(M) = \begin{cases} V & \text{if } M \mapsto_d^* V \\ \bot & \text{if } \forall j \in \mathbb{N}, M_j \mapsto_d M_{j+1}, \text{with } M_0 = M \\ \mathsf{error} & \text{if } M \mapsto_d^* M_s, \text{with } M_s \in \mathit{Stuck}(\Lambda_d) \end{cases}$$

Dynamically Bound Variables:

$$\begin{aligned}
DBV([\,]) &= \emptyset \\
DBV(V\ \mathcal{E}) &= DBV(\mathcal{E}) \\
DBV(\mathcal{E}\ M) &= DBV(\mathcal{E}) \\
DBV(\mathsf{dlet}\ (x_d\ V)\ \mathcal{E}) &= \{x_d\} \cup DBV(\mathcal{E})
\end{aligned}$$

Stuck Terms:
$M \in \mathit{Stuck}(\Lambda_d)$ if
$M = \mathcal{E}[x_d]$ with $x_d \notin DBV(\mathcal{E})$

Fig. 6. Sequential Evaluation Function

Intuitively, the value of a dynamic variable is given by the latest active binding for this variable. In this framework, the latest active binding corresponds to the innermost dlet that binds this variable. The dynamic extent of a dlet construct is the period of time between its apparition by (dlet *intro*) and its elimination by (dlet *elim*).

The evaluation algorithm introduces the concept of *stuck term*, which is defined by the occurrence of a dynamic variable in an evaluation context that does not contain a binding for it. The evaluation function is then defined as a total function returning a value when evaluation terminates, $\bot$ when evaluation diverges, or error when a stuck term is reached.

The correctness of the evaluation function is established by the following Theorem, which relates eval_c and eval_d. Let us observe that eval_c may return a value V' that differs from the value V returned by eval_d because the calculus can perform reductions inside abstractions.

Theorem 2 *For any program $M \in \Lambda_d^0$,* $\mathrm{eval}_c(M) = V'$ *iff* $\mathrm{eval}_d(M) = V$. □

If we were to implement (*lookup*), we would start from the dynamic variable to be evaluated, and search for the innermost enclosing dlet. If it contained a binding for the variable, we would return the associated value. Otherwise, we would proceed with the next enclosing dlet. This behaviour exactly corresponds to the search of a value in an associative list (`assoc` in Scheme). Such a strategy is usually referred to as *deep binding*. In Section 7, we further refine the sequential evaluation function by making this associative list explicit. But, beforehand, we show that dynamic binding adds expressiveness to a functional language.

5 Expressiveness

In Section 2.1, we stated that dynamic binding was an expressive programming technique that, when used in a sensible manner, could reduce programming patterns in programs. In this Section, we give a formal justification to this statement, by proving that dynamic binding adds expressiveness [8] to a purely functional language. First, we define the notion of observational equivalence.

Definition 7 (Observational Equivalence) Given a programming language $\mathcal{L}$ and an evaluation function $eval_{\mathcal{L}}$, two terms $M_1, M_2 \in \mathcal{L}$ are *observationally equivalent*, written $M_1 \cong_{\mathcal{L}} M_2$, if for any context $C \in \mathcal{L}$, such that $C[M_1]$ and $C[M_1]$ are both programs of $\mathcal{L}$, $eval_{\mathcal{L}}(M_1)$ is defined and equal to V if and only if $eval_{\mathcal{L}}(M_2)$ is defined and equal to V. □

We shall denote the observational equivalences for the call-by-value λ-calculus and for the λ_d-calculus by $\cong_v$ and $\cong_d$, respectively. In order to prove that dynamic binding adds expressiveness [8] to a purely functional language, let us consider the following lambda terms, assuming the existence of a primitive cons to construct pairs.

$$M_1 = \lambda tf.(\mathsf{cons}\ (t\ 0)\ (f\ (\lambda d.(t\ 0)))) \quad M_2 = \lambda tf.(\mathsf{let}\ (v\ (t\ 0))\quad (\mathsf{cons}\ v\ (f\ (\lambda d.v))))$$

The terms M_1, M_2 are observationnally equivalent in the λ_v-calculus, i.e. $M_1 \cong_v M_2$, but we have that $M_1 \not\cong_d M_2$. Indeed, if $C \in \Lambda_d$ is $C = (\lambda x_d.\ ([\,]\ (\lambda d.x_d)\ (\lambda t.\ (\lambda x_d.\ (t\ 0))\ 1)))\ 0$, then $C[M_1] = (\mathsf{cons}\ 0\ 1)$, while $C[M_2] = (\mathsf{cons}\ 0\ 0)$.

This example shows that dynamic binding enables us to distinguish terms that the call-by-value λ-calculus cannot distinguish. As a result, $\cong_v \not\subset \cong_d$, and using Felleisen's definition of expressiveness [8, Thm 3.14], we conclude that:

Proposition 1. *Λ_v cannot macro-express dynamic binding relative to Λ_d.*

6 Semantics of Exceptions

First-class continuations and state can simulate exceptions [13]. We show here that exceptions can be defined in terms of first-class continuations and dynamic binding.

In the semantics of ML [26], a raised exception returns an *exceptional* value, distinct from a *normal* value, which has the effect to prune its evaluation context until a handler is able to deal with the exception. By merging the mechanism that aborts the computation and the mechanism that fetches the handler for the exception, the handler can no longer be executed in the dynamic environment in which the exception was raised. As a result, such an approach cannot be used to give a semantics to other kinds of exceptions, like resumable ones [43].

In order to model the abortive effect, we extend the sequential evaluation function of Figure 6 with Felleisen and Friedman's abort operator $\mathcal{A}$ [11]. For the sake of simplicity, we assume that there exists only one exception type (discrimination on the kind of exception can be performed in the handler). We also assume the existence of a distinguished dynamic variable x_{ed}. In Figure 7, we give the semantics of ML-style exceptions. When an exception is raised, the latest active handler is called, escapes, and then applies f in the same dynamic environment as handle, and not in the dynamic environment where the exception was raised[3].

On the other hand, there exist other kinds of exceptions, like resumable exceptions, e.g. Common Lisp resumable errors [43], or Eulisp resumable conditions [34]. They essentially offer the opportunity to resume the computation at the point where the exception was raised. In the sequel, we present a variant of Queinnec's *monitors* [36,

[3] The usage of a first-class continuation appears here as the rule for handle duplicates the evaluation context $\mathcal{E}$. Let us also observe that the continuation is only used in a downward way, which amounts to popping frames from the stack only.

$M \in \Lambda_d ::= \ldots \mid (\mathcal{A}\ M)$ (Term) $\quad \mathcal{E}[(\mathsf{handle}\ f\ M)] \mapsto_d \mathcal{E}[(\lambda x_{ed}.M)\ (\lambda v.\mathcal{A}\ \mathcal{E}[(f\ v)])]$

$\mathcal{E}[\mathcal{A}\ M] \mapsto_d M$ (Abort) $\quad \mathcal{E}[(\mathsf{raise}\ V)] \mapsto_d \mathcal{E}[(x_{ed}\ V)]$

Fig. 7. ML-style exceptions

p. 255], which give the essence of resumable exceptions. The primitives monitor/signal play the role that handler/raise had for ML-style exceptions. Let us note that signal is a binary function, which takes not only a value, but also a boolean r indicating whether the exception should be raised as resumable.

$$\mathcal{E}[(\mathsf{monitor}\ f\ M)] \mapsto_d \mathcal{E}[(\lambda x_{ed}.M)\,(\mathsf{let}\ (old\ x_{ed})\ \ (\lambda\ r\ v.\ \ (\mathsf{let}\ (x\ ((\lambda x_{ed}.(f\ r\ v))\ old))\ (\mathbf{if}\ r\ x\ (\mathcal{A}\ \mathcal{E}[x])))))]$$

$$\mathcal{E}[(\mathsf{signal}\ r\ V)] \mapsto_d \mathcal{E}[(x_{ed}\ r\ V)]$$

Fig. 8. Resumable exceptions

Like handle, monitor installs an exception handler for the duration of a computation. If an exception is signalled, the latest active handler is called in the dynamic environment of the signalled exception. If an exception is signalled by the handler itself, it will be handled by the handler that existed *before* monitor was called: this is why x_{ed} is shadowed for the duration of the execution of the handler f, but will be again accessible if the "normal" computation resumes. If the exception was signalled as *resumable*, i.e. if the first argument of signal is true, the value returned by the handler is returned by signal, and computation continues in exactly the same dynamic environment[4].

This approach to define the semantics of exception has two advantages, at least. First, as we model each *effect* by the appropriate primitive (abortion by $\mathcal{A}$ and handler installation by dynamic binding), we have the ability to model different kinds of semantics for exceptions. Second, defining the semantics of exceptions with assignments weakens the theory [12] because assignments break some equivalences that would hold in the presence of exceptions: so, our definition provides a more precise characterisation of a theory of exceptions.

7 Refinement

We refine the evaluation function by representing the dynamic environment explicitly by an associative list. By separating the evaluation context from the dynamic environment, we facilitate the design of a parallel evaluation function of Section 8.

Figure 9 displays the state space and transition rules of the deep binding strategy. The dynamic environment is represented in a new dlet construct which can only appear at the outermost level of a configuration, called state. The list of bindings δ can be regarded as a global stack, initially empty when evaluation starts. A binding is pushed on the binding list, every time a dynamic abstraction is applied, and popped at the end of the dynamic extent of the application. In Section 4, the dlet construct was also modelling the dynamic extent of a dynamic-abstraction application; now that the dlet construct no longer appears inside terms, we introduce a (pop M) term playing the same role: it is created when a dynamic abstraction is applied and is destroyed at the end of the dynamic extent, after popping the top binding of the binding list. Theorem 3 establishes the correctness of the deep binding strategy.

[4] Such a semantics assumes that there exists an initial handler in which evaluation can proceed.

State Space:

$$\begin{array}{llll}
S \in State_{db} & ::= & (\mathsf{dlet}\ \delta\ M) & (State) \\
M \in \Lambda_{db} & ::= & V \mid x_d \mid (M\ M) \mid (\mathsf{pop}\ M) & (Term) \\
V \in Value_{db} & ::= & x_s \mid (\lambda x_s.M) \mid (\lambda x_d.M) & (Value) \\
\delta \in Bind_{db} & ::= & () \mid \delta\ \S\ ((x_d\ V)) & (Binding\ list) \\
x_s \in SVar & = & \{x_{s0}, x_{s1}, \ldots\} & (Static\ Variable) \\
x_d \in DVar & = & \{x_{d0}, x_{d1}, \ldots\} & (Dynamic\ Variable) \\
\mathcal{E} \in EvCon_{db} & ::= & [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{pop}\ \mathcal{E}) & (Evaluation\ Context)
\end{array}$$

Transition Rules:

$$\begin{array}{rlr}
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda x_s.M)\ V]) & \mapsto_{db} (\mathsf{dlet}\ \delta\ \mathcal{E}[M\{V/x_s\}]) & (\beta_v) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda x_d.M)\ V]) & \mapsto_{db} (\mathsf{dlet}\ \delta\S((x_d\ V))\ \mathcal{E}[(\mathsf{pop}\ M)]) & (\mathsf{dlet}\ extend) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[x_d]) & \mapsto_{db} (\mathsf{dlet}\ \delta\ \mathcal{E}[V]) \ \text{ if }\ V = \mathrm{lk}(x_d, \delta) & (lookup) \\
(\mathsf{dlet}\ \delta\S((x_d\ V))\ \mathcal{E}[(\mathsf{pop}\ V')]) & \mapsto_{db} (\mathsf{dlet}\ \delta\ \mathcal{E}[V']) & (pop)
\end{array}$$

Evaluation Function:

$$\forall M \in \Lambda_u^0,\ \mathrm{eval}_{db}(M) = \begin{cases} V & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_{db}^{*} (\mathsf{dlet}\ ()\ V) \\ \bot & \text{if } \forall j \in \mathbb{N}, M_j \mapsto_{db} M_{j+1}, \text{with } M_0 = (\mathsf{dlet}\ ()\ M) \\ \mathsf{error} & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_{db}^{*} M_s, \text{with } M_s \in Stuck(\Lambda_{db}) \end{cases}$$

Stuck State: $S \in Stuck(\Lambda_{db})$,
if $S = (\mathsf{dlet}\ \delta\ \mathcal{E}[x_d])$ with $x_d \notin DOM(\delta)$

$$\mathrm{lk}(x_d, \delta\S((x_d\ V))) = V$$
$$\mathrm{lk}(x_d, \delta\S((x_{d1}\ V))) = \mathrm{lk}(x_d, \delta) \text{ if } x_d \neq x_{d1}$$

Fig. 9. Deep Binding

Theorem 3 $\mathrm{eval}_d = \mathrm{eval}_{db}$ □

The deep binding technique is simple to implement: bindings are pushed on the binding list δ at application time of dynamic abstractions and popped at the end of their extent. However, the lookup operation is inefficient because it requires searching the dynamic list, which is an operation linear in its length.

There exist some techniques to improve the lookup operation. The *shallow binding* technique consists in indexing the dynamic environment by the variable names [1]. A further optimisation, called *shallow binding with value cell* is to associate each dynamic variable with a fixed location which contains the correct binding for that variable: the lookup operation then simply requires to read the content of that location.

8 Parallel Evaluation

In Section 3, we observed that the axiom (dlet *propagate'*) was particularly suitable for parallel evaluation because it allowed the independent evaluation of the operator and operand by duplicating the dynamic environment. It is well-known that the deep binding strategy is adapted to parallel evaluation because the associative list representing the dynamic environment can be shared between different tasks.

As in our previous work [30], we follow the "parallelism by annotation" approach, where the programmer uses an annotation future [17] to indicate which expressions may be evaluated in parallel. The semantics of future has been described in the purely functional framework [14] and in the presence of first-class continuations and assignments [30]. In Figure 10, we present the semantics of future in the presence of dynamic binding.

As in [14, 30], the set of terms is augmented with a future construct, and we add to the set of values a placeholder variable, "which represents the result of a computation that is in progress". In addition, a new construct (f-let $(p\ M)\ S$) has a double goal: first as a let, it binds p to the value of M in S; second, it models the potential evaluation of S in parallel with M. The component M is the *mandatory* term because it is the first that would be evaluated if evaluation was sequential, while S is *speculative* because its value is not known to be needed before M terminates.

State Space:

$$\begin{array}{lll r}
S \in State_p & ::= & (\text{dlet}\ \delta\ M)\ \mid\ (\text{dlet}\ \delta\ (\text{f-let}\ (p\ M)\ S))\ \mid\ \text{error} & (State)\\
M \in \Lambda_p & ::= & V\ \mid\ x_d\ \mid\ (M\ M)\ \mid\ (\text{future}\ M) & (Term)\\
 & & \mid (\text{pop}\ M)\ \mid\ (\text{fmark}\ \delta\ M)\ \mid\ (\mathcal{A}\ \text{error}) & \\
W \in PValue_p & ::= & x_s\ \mid\ (\lambda x_s.M)\ \mid\ (\lambda x_d.M) & (Proper\ Value)\\
V \in Value_p & ::= & W\ \mid\ p & (Runtime\ Value)\\
g \in AValue & ::= & f\ \mid\ (\lambda x_s.M)\ \mid\ (\lambda x_d.M) & (Applicable\ Value)\\
\mathcal{D} \in SeqEvCon_p & ::= & [\,]\ \mid\ (V\ \mathcal{D})\ \mid\ (\mathcal{D}\ M)\ \mid\ (\text{pop}\ \mathcal{D})\ \mid\ (\text{fmark}\ \delta\ \mathcal{D}) & (Seq.\ Ev.\ Context)\\
\mathcal{E} \in EvCon_p & ::= & \mathcal{D}\ \mid\ (\text{f-let}\ (p\ \mathcal{D})\ S) & (Ev.\ Context)
\end{array}$$

Transition Rules:

$$\begin{array}{rl r}
(\text{dlet}\ \delta\ \mathcal{E}[V_1\ V_2]) \mapsto_p^{1,1} & \begin{cases}(\text{dlet}\ \delta\ \mathcal{E}[M\{V_2/x_s\}]) & \text{if } V_1 = (\lambda x_s.M)\\ (\text{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \text{error})]) & \text{if } V_1 \notin AValue, V_1 \neq p\end{cases} & (\beta_v)\\
(\text{dlet}\ \delta\ \mathcal{E}[(\lambda x_d.M)\ V]) \mapsto_p^{1,1} & (\text{dlet}\ \delta\S((x_d\ V))\ \mathcal{E}[(\text{pop}\ M)]) & (\text{dlet}\ extend)\\
(\text{dlet}\ \delta\ \mathcal{E}[x_d]) \mapsto_p^{1,1} & \begin{cases}(\text{dlet}\ \delta\ \mathcal{E}[V]) & \text{if } V = \delta(x_d)\\ (\text{dlet}\ \delta\ \mathcal{E}[\mathcal{A}\ \text{error}]) & \text{if } x_d \notin DOM(\delta)\end{cases} & (lookup)\\
(\text{dlet}\ \delta\S((x_d\ V))\ \mathcal{E}[(\text{pop}\ V')]) \mapsto_p^{1,1} & (\text{dlet}\ \delta\ \mathcal{E}[V']) & (pop)\\
(\text{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \text{error})]) \mapsto_p^{1,1} & \text{error} & (error)\\
(\text{dlet}\ \delta\ \mathcal{E}[(\text{future}\ M)]) \mapsto_p^{1,1} & (\text{dlet}\ \delta\ \mathcal{E}[(\text{fmark}\ \delta\ M)]) & (ltc)\\
(\text{dlet}\ \delta\ \mathcal{E}[(\text{fmark}\ \delta_1\ V)]) \mapsto_p^{1,1} & (\text{dlet}\ \delta_1\ \mathcal{E}[V]) & (future\ id)\\
(\text{dlet}\ \delta\ \mathcal{E}[(\text{fmark}\ \delta_1\ M)]) \mapsto_p^{1,0} & (\text{dlet}\ \delta(\text{f-let}\ (p\ M)(\text{dlet}\ \delta_1\mathcal{E}[p])))\ p \notin FP(\mathcal{E}) \cup FP(\delta_1) & (fork)\\
(\text{dlet}\ \delta\ (\text{f-let}\ (p\ V)\ S)) \mapsto_p^{1,1} & S\{V/p\} & (join)\\
(\text{dlet}\ \delta\ (\text{f-let}\ (p\ M)\ S_1)) \mapsto_p^{1,0} & (\text{dlet}\ \delta\ (\text{f-let}\ (p\ M)\ S_2))\ \text{ if } S_1 \mapsto_p^{1,1} S_2 & (speculative)\\
S \mapsto_p^{0,0} & S & (reflexive)\\
S \quad \mapsto_p^{a+a',b+b'} & S'' \text{ if } S \mapsto_p^{a,b} S' \text{ and } S' \mapsto_p^{a',b'} S''. & (transitive)
\end{array}$$

Evaluation Function: For any program $M \in \Lambda_u^0$,

$$\text{eval}_p(M) = \begin{cases} W & \text{if } (\text{dlet}\ ()\ M) \mapsto_p^{*} (\text{dlet}\ ()\ W)\\ \bot & \text{if } \forall j \in \mathbb{N}, \exists n_j, m_j \in \mathbb{N} \text{ such that}\\ & \quad (\text{dlet}\ ()\ M) = S_0 \text{ and } S_j \mapsto_p^{n_j,m_j} S_{j+1} \text{ with } m_j > 0.\\ \text{error} & \text{if } (\text{dlet}\ ()\ M) \mapsto_p^{*} M_s, \text{with } M_s \in Stuck(\Lambda_{db}), \text{ or } (\text{dlet}\ ()\ M) \mapsto_p^{*} \text{error}\end{cases}$$

Fig. 10. Parallel Evaluation (differences with Figure 9)

It is important to observe that (future [ ]) is not a valid evaluation context. Otherwise, if evaluation was allowed to proceed inside the future body, it could possibly change the dynamic environment, which would make (*fork*) unsound. Instead, rule (*ltc*), which stands for *lazy task creation* [27, 7], replaces a (future M) expression by (fmark δ M), which should be interpreted as a mark indicating that a task may be created.

If the runtime elects to create a new task, (*fork*) creates a f-let expression, whose mandatory component is the argument of fmark, i.e. the future argument, and whose speculative component is a new state evaluating the context of fmark filled with the placeholder variable, in the scope of the duplicated dynamic environment δ_1. If the run-time does not elect to spawn a new task, evaluation can proceed in the fmark argument.

Rules (*ltc*) and (*future id*) specify the sequential behaviour of future: the value of future is the value of fmark, which is the value of its argument.

When the evaluation of the mandatory component terminates, rule (*join*) substitutes the value of the placeholder in the speculative state. Rule (*speculative*) indicates that speculative transitions are allowed in the f-let body.

Following [14], Figure 10 defines a relation $S_1 \mapsto_p^{n,m} S_2$ meaning that n steps are involved in the reduction from S_1 to S_2, among which m are mandatory.

The correctness of the evaluation function follows from a modified diamond property and by the observation that the number of pop terms in a state is always smaller or equal to the length of the dynamic environment.

Theorem 4 $\text{eval}_{db} = \text{eval}_p$ □

As far as implementation is concerned, rule (*ltc*) seems to indicate that the dynamic environment should be duplicated. A further refinement of the system indicates that it suffices to duplicate a *pointer* to the associative list, as long as the list remains accessible in a shared store.

Rule (*ltc*) adds an overhead to every use of future, by duplicating the dynamic environment even if dynamic variables are not used. Feeley [7] describes an implementation that avoids this cost by lazily recreating a dynamic environment when a task is stolen.

Due to the orthogonality between assignments and dynamic binding, our previous results [30] with assignments can be merged within this framework. Adding assignments permits the definition of mutable dynamic variables (with a construct like `dynamic-set!` [34]). Due to the purely dynamic nature of the semantics, the presence of *mutable* dynamic variables offers less parallelism as observed in [30]. The interaction of dynamic binding and continuations is however beyond the scope of this paper [19].

9 Related Work

In the conference on the History of Programming Languages, McCarthy [25] relates that they observed the behaviour of dynamic binding on a program with higher-order functions. The bug was fixed by introducing the funarg device and the `function` construct[32].

Cartwright [4] presents an equational theory of dynamic binding, but his language is extended with explicit substitutions and assumes a call-by-name parameter passing technique. The motivation of his work fundamentally differs from ours: his goal is to derive a homomorphic model of functional languages by considering λ as a combinator. His axioms are derived from the $\lambda\sigma$-calculus axioms, while ours are constructed during the proof of equational correspondence of the calculus.

The authors of [6] discuss the issue of tail-recursion in the presence of dynamic binding. They observe that simple implementations of `fluid-let` [18] are not tail-recursive because they restore the previous dynamic environment after evaluating the `fluid-let` body. Therefore, they propose an implementation strategy, which in essence is a dynamic-environment passing style solution. Programs in dynamic-environment passing style are characterised by the fact that they do not require a growth of the control state for dynamic binding; however, they require a growth of the heap space. An analogy is the continuation-passing translation, which generates a program where all function calls are in terminal position although it does not mean that all cps-programs are iterative. Feeley [7] and Queinnec [36] observe that programs in dynamic-environment passing style reserve a special register for the current dynamic environment. Since *every* non-terminal call saves and then restores this register, such a strategy penalises programs that do not use dynamic binding, especially in byte-code interpreters where the marginal cost of an extra register is very high. Both of them prefer a solution that does not penalise all programs, at the price of a growth of the control state for every dynamic binding. Consequently, we believe that implementors have to decide whether dynamic binding should or not increase the control state; in any case, it will result in a non-iterative behaviour.

10 Conclusion

In the tradition of the syntactic theories for continuations and assignments, we present a syntactic theory of dynamic binding. This theory helps us in deriving a sequential evaluation function and a refined implementation like deep binding. We also integrate dynamic-binding constructs into our framework for parallel evaluation of future-based programs.

Besides, we prove that dynamic binding adds expressiveness to purely functional language and we show that dynamic binding is a suitable tool to define the semantics of exceptions-like notions. Furthermore, we believe that a single framework integrating continuations, side-effects, and dynamic binding would help us in proving implementation strategies of `fluid-let` in the presence of continuations [19].

11 Acknowledgement

Many thanks to Daniel Ribbens, Christian Queinnec, and the anonymous referees for their helpful comments.

References

1. John Allen. *Anatomy of Lisp.* Mc Graw Hill, 1979.
2. Henry Baker. Shallow binding in lisp 1.5. *Comm. of the ACM*, 21(7):565–569, 1978.
3. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.
4. Robert Cartwright. Lambda: the Ultimate Combinator. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 27–46. Academic Press, 1991.
5. Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, June 1990.
6. Bruce F. Duba, Matthias Felleisen, and Daniel P. Friedman. Dynamic Identifiers Can Be Neat. Technical Report 220, Indiana University, Computer Science Department, 1987.
7. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, 1993.
8. Matthias Felleisen. On the Expressive Power of Programming Languages. In *Proc. European Symposium on Programming*, in LNCS 432, pages 134–151. Springer-Verlag, 1990.
9. Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *Parallel Architecture and Languages Europe*, in LNCS 259, pages 206–223, 1987.
10. Matthias Felleisen and Daniel P. Friedman. A Syntactic Theory of Sequential State. *Theoretical Computer Science*, 69:243–287, 1989.
11. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theoretical Computer Science*, 52(3):205–237, 1987.
12. Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992.
13. Andrzej Filinski. *Controlling Effects.* PhD thesis, School of Computer Science. Carnegie Mellon University, May 1996.
14. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995.
15. Michael J.C. Gordon. Operational Reasoning and Denotational Semantics. In *Proving and Improving Programs*, Colloques IRIA, pages 83–98, Arc et Senans, July 1975.
16. Michael J.C. Gordon. Towards a Semantic Theory of Dynamic Binding. Technical Report STAN-CS-75-507, Stanford University, August 1975.

17. Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In *Parallel Lisp : Languages and Systems*, in LNCS 441, pages 2–57. Springer-Verlag, 1990.
18. Chris Hanson. *MIT Scheme Reference Manual.* Massachusetts Inst. of Tech., Jan. 1991.
19. Christopher Haynes and Daniel P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, 1987.
20. Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, 1990.
21. Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). *Report on the Programming Language Haskell.* 1991.
22. Donald E. Knuth. *The TEXbook.* Addison-Wesley, 1994.
23. Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman, and Chris Welt. *GNU Emacs Lisp Reference Manual*, 2.4 edition.
24. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
25. John McCarthy. History of Lisp. In *ACM SIGPLAN History of Programming Languages Conference*, ACM Monograph Series, pages 173–196, June 1978.
26. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.
27. Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation : a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.
28. David A. Moon. Maclisp reference manual. Technical report, MIT Project Mac, April 1974.
29. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* PhD thesis, University of Liège, Liège, Belgium, June 1994.
30. Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 146–156, May 1996.
31. Luc Moreau and Christian Queinnec. Partial Continuations as the Difference of Continuations. A Duumvirate of Control Operators. In *International Conference on Programming Language Implementation and Logic Programming (PLILP'94)*, in LNCS 844, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.
32. Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. Project MAC AI-199, M.I.T., June 1970.
33. Randy B. Osborne. Speculative Computation in Multilisp. In *Parallel Lisp : Languages and Systems*, in LNCS 441, pages 103–137. Springer-Verlag, 1990.
34. Julian Padget and Grep Nuyens (Editors). The Eulisp Definition, June 1991.
35. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.
36. Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996. ISBN 0 521 56247 3.
37. Christian Queinnec and David De Roure. Design of a Concurrent and Distributed Language. In *Parallel Symbolic Computing: Languages, Systems and Applications*, in LNCS 748, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.
38. Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 174–184, 1991.
39. Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.
40. Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: a Synthesis of Two Paradigms.* PhD thesis, Rice University, 1994.
41. Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic and Computation*, 6(3/4):289–360, November 1993.
42. Dorai Sitaram and Matthias Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.
43. Guy Lewis Steele, Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.
44. Carolyn Talcott. Rum : an Intensional Theory of Function and Control Abstractions. In *Proc. 1987 Workshop on Foundations of Logic and Functional Programming*, in LNCS 306, pages 3–44. Springer-Verlag, 1988.
45. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., second edition edition, 1996.